CHAPTER 16

# PROGRAMMING LANGUAGES AND CODING

All the software engineering steps that have been presented to this point are directed toward a final objective: to translate representations of software into a form that can be "understood" by the computer. We have (finally) reached the coding step—a process that transforms design into a programming language. Gerald Weinberg [WEI71] expressed the true meaning of coding when he wrote: "...when we talk to our computers, unhappily, we are usually speaking in different tongues...."

Most readers of this book may live to see the day when the above quotation is proved incorrect. Requests for computer processing services may be coded (or spoken) in a natural language, such as English. Already, a set of so-called *fourth-generation techniques* is changing our understanding of the term "programming language." Rather than coding, developers of some classes of management information systems (and limited areas of engineering and scientific applications) can now describe desired results, rather than desired procedure, in a *nonprocedural language*. Conventional programming language source code is then automatically generated.

However, the vast majority of software applications still reside beyond the reach of fourth-generation approaches. For the time being, we code using artificial languages such as Ada, FORTRAN, PASCAL, C, COBOL, or assembler language.

When considered as a step in the software engineering process, coding is viewed as a natural consequence of design. However, programming language characteristics and coding style can profoundly affect software quality and maintainability. This chapter does not aspire to teach the reader to code.

Rather, topics associated with programming languages and coding are presented in the broader context of software engineering.

## 16.1 THE TRANSLATION PROCESS

The coding step translates a detail design representation of software into a programming language realization. The translation process continues when a compiler accepts *source code* as input and produces machine-dependent *object code* as output. Compiler output is further translated into *machine code*—the actual instructions that drive microcoded logic in the central processing unit.

The initial translation step—from detail design to programming language—is a primary concern in the software engineering context. "Noise" can enter the translation process in many ways. Improper interpretation of a detail design specification can lead to erroneous source code [SHE81]. Programming language complexity or restrictions can lead to convoluted source code that is difficult to test and maintain. More subtly, characteristics of a programming language can influence the way we think, propagating unnecessarily limited software designs and data structures.

For example, a design directed at a target FORTRAN 77 implementation would be less likely to select a linked list data structure, because FORTRAN 77 does not directly support such a structure. If the target language were C or PASCAL (both languages provide direct support for linked lists), the linked list would be a more feasible alternative.

Language characteristics have an impact on the quality and efficiency of translation. In the next section, we evaluate language characteristics by considering two different views of programming languages.

## 16.2 PROGRAMMING LANGUAGE CHARACTERISTICS

Programming languages are vehicles for communication between humans and computers. The coding process—communication via a programming language—is a human activity. As such, the psychological characteristics of a language have an important impact on the quality of communication. The coding process may also be viewed as one step in the software engineering process. The engineering characteristics of a language have an important impact on the success of a software development project. Finally, technical characteristics of a language can influence the quality of design (recall that practicality often dictates that detail design be directed toward a specific programming language). Therefore, technical characteristics can affect both human and software engineering concerns.

### 16.2.1 A Psychological View

In his book *Software Psychology*, Ben Shneiderman [SHN80] observed that the role of the software psychologist is to "focus on human concerns such as ease of use, simplicity in learning, improved reliability, reduced error frequency and enhanced user satisfaction, while maintaining an awareness of machine efficiency, software capacity, and hardware constraints." Even though automated tools (CASE) provide substantial assistance, software engineering remains an intensely human activity. We still have much to learn about the human aspects of computer-based system development.

Another software psychologist, Gerald Weinberg [WEI71], relates a story that bears repeating (in paraphrased form) when we consider characteristics of programming languages:

> It is impossible to begin a discussion of psychological principles of programming language design without recalling the story of "The Genius Tailor." It seems that a man had gone to the tailor to have a suit made cheaply, but when the suit was finished and he went to try it on, it didn't fit him at all.
>
> Complaining that the jacket was too big in back, the right arm was too long, one pant leg was too short and three buttons were missing, the man was justifiably upset.
>
> "No problem," said the tailor, "just hunch your back, bend your arm, walk with a limp, and stick your fingers through the button holes and you'll look just fine!"
>
> The man contorted his body to fit the suit and feeling duped by the tailor, he left. He had not walked one block when he was approached by a stranger.
>
> "Who made that suit for you?" asked the stranger. "I'm in the market for a new suit myself."
>
> Surprised, but pleased at the compliment, the man pointed out the tailor's shop.
>
> "Well, thanks very much," said the stranger, hurrying off. "I do believe I'll go to that tailor for my suit. Why, he must be a genius to fit a cripple like you!"

Weinberg suggests that we could extend this parable to a story of the genius programming language designer. The designers of programming languages often make us contort our approach to a problem so that the approach will fit the constraints imposed by a specific programming language. Because human factors are critically important in programming language design, the psychological characteristics of a language have a strong bearing on the success of design to code translation and implementation.

A number of psychological characteristics [WEI71] occur as a result of programming language design. Although these characteristics are not measurable in any quantifiable way, we recognize their manifestation in all programming languages. We discuss each characteristic briefly in the paragraphs that follow.

*Uniformity* indicates the degree to which a language uses consistent notation, applies seemingly arbitrary restrictions, and supports syntactic or semantic exceptions to the rule. For example, FORTRAN uses the parentheses as delimiter for array indices, as a modifier for arithmetic precedence, and as a delimiter for a subprogram argument list (to name a few!). This multiuse notation has led to more than a few subtle errors.

*Ambiguity* in a programming language is perceived by the programmer. A compiler will always interpret a statement in one way, but the human reader may interpret the statement differently. Here lies psychological ambiguity. For example, psychological ambiguity arises when arithmetic precedence is not obvious:

```
X = X1/X2*X3
```

One reader of the source code might interpret the above as $X = (X_1/X_2)*X_3$ while another reader might "see" $X = X_1/(X_2*X_3)$. Another potential source of ambiguity is the nonstandard use of identifiers that have default data types. For example, in FORTRAN an identifier KDELTA would be assumed (by default) to have *integer* characteristics. However, an explicit declaration, REAL KDELTA, could cause confusion due to psychological ambiguity.

A lack of uniformity and the occurrence of psychological ambiguity normally occur together. If a programming language exhibits the negative aspects of these characteristics, source code is less readable and translation from design is more error-prone.

*Compactness* of a programming language is an indication of the amount of code-oriented information that must be recalled from human memory. Among the language attributes that measure compactness are:

- The degree to which a language supports the structured constructs (Chapter 10) and logical "chunking"
- The kinds of keywords and abbreviations that may be used
- The variety of data types and default characteristics
- The number of arithmetic and logical operators
- The number of built-in functions.

APL is an exceptionally compact programming language. Its powerful and concise operators allow relatively little code to accomplish significant arithmetic and logical procedures. Unfortunately, the compactness of APL also makes the language difficult to read and understand, and can lead to poor uniformity (e.g., the use of *monadic* and *dyadic* forms for the same operator symbol).

The characteristics of human memory have a strong impact on the way in which we use language. Human memory and recognition may be divided into *synesthetic* and *sequential* domains [KLA80]. Synesthetic memory allows us to remember and recognize things as a whole. For example, we recognize a human face instantly; we do not consciously evaluate each of its distinct parts prior to recognition. Sequential memory provides a means for recalling the next element in a sequence (e.g., the next line in a song, given the preceding lines). Each of these memory characteristics affect programming language characteristics that are called locality and linearity.

*Locality* is the synesthetic characteristic of a programming language. Locality is enhanced when statements may be combined into *blocks*, when the structured constructs may be implemented directly, and when design and resultant code are highly modular and cohesive (Chapter 10). A language characteristic that supports or encourages exception handling (e.g., ON-condition processing in PL/1 or ERR = in extended versions of FORTRAN) violates locality.

*Linearity* is a psychological characteristic that is closely associated with the concept of maintenance of functional domain. That is, human perception is facilitated when a linear sequence of logical operations is encountered. Extensive branching (and, to some extent, large loops) violates the linearity of processing. Again, direct implementation of the structured constructs aids programming language linearity.

Our ability to learn a new programming language is affected by *tradition*. A software engineer with a background in FORTRAN or ALGOL would have little difficulty learning PL/1, PASCAL, or C. The latter languages have a tradition established by the former. Constructs are similar, form is compatible, and a sense of programming language "format" is maintained. However, if the same individual were required to learn APL , LISP, or Smalltalk, tradition would be broken and time on the learning curve would be longer.

Tradition also affects the degree of innovation during the design of a new programming language. Although new languages are proposed frequently, new language forms evolve slowly. For example, PASCAL is a close relative of ALGOL. However, a major innovation in the PASCAL language [JEN74] is an implementation of user-defined data types, a form that does not exist in earlier languages tied to PASCAL by tradition. Ada, a language that has also grown out of the ALGOL-PASCAL tradition, extends beyond both languages with a wide variety of innovative structures and typing.

The psychological characteristics of programming languages have an important bearing on our ability to learn, apply, and maintain them. In summary, a programming language colors the way we think about programs and inherently limits the way in which we communicate with a computer. Whether this is good or bad remains an open question.

### 16.2.2 A Syntactic/Semantic Model

Shneiderman [SHN80] has developed a *syntactic-semantic model* of the programming process that has relevance in a consideration of the coding step. When a programmer applies software engineering methods (e.g., requirements analysis, design) that are programming language-independent, semantic knowledge is tapped. Syntactic knowledge, on the other hand, is language-dependent, concentrating on the characteristics of a specific language.

Of these knowledge types, semantic knowledge is the more difficult to acquire and the more intellectually demanding to apply. All software engineering steps that precede coding make heavy use of semantic knowledge. The coding step applies syntactic knowledge that is "arbitrary and instructional" and learned by rote [SHN80]. When a new programming language is learned, new syntactic information is added to memory. Potential confusion may occur when the syntax of a new programming language is similar but not equivalent to the syntax of another language. It should be noted, however, that a new programming language can serve to force the software engineer to learn new semantic information as well. For example, the Ada programming language has caused many software engineers to rethink their approach to the design and implementation of software-based systems.

When arguments about the compelling need to "generate code" arise, the listener should realize that many problems associated with computer software have not been caused by a lack of syntactic knowledge. The problem lies in the scope of our semantic knowledge and our ability to apply it. The goal of software engineering is to expand the knowledge of the semantics of software development.

### 16.2.3 An Engineering View

A software engineering view of programming language characteristics focuses on the needs of a specific software development project. Although esoteric requirements for source code may be derived, a general set of engineering characteristics can be established: (1) ease of design to code translation, (2) compiler efficiency, (3) source code portability, (4) availability of development tools, and (5) maintainability.

The coding step begins after a detail design has been defined, reviewed, and modified, if necessary. In theory, source code generation from a detail design specification should be straightforward. *Ease of design to code translation* provides an indication of how closely a programming language mirrors a design representation. As we discussed in Section 16.1, a language that directly implements the structured constructs, sophisticated data structures, specialized I/O, bit manipulation capabilities, and object-oriented constructs will make translation from design to source code much easier (if these attributes are specified in the design).

Although rapid advances in processor speed and memory density have begun to mitigate the need for "superefficient code," many applications still require fast, "tight" (low memory requirement) programs. An on-going criticism of high-level language compilers is directed at an inability to produce fast, tight, executable code. Languages with optimizing compilers may be attractive if software performance is a critical requirement.

*Source code portability* is a programming language characteristic that may be interpreted in three different ways:

1. Source code may be transported from processor to processor and compiler to compiler with little or no modification.
2. Source code remains unchanged even when its environment changes (e.g., a new version of an operating system is installed).
3. Source code may be integrated into different software packages with little or no modification required because of programming language characteristics.

Of the three interpretations of portability, the first is by far the most common. Standardization (by the International Standards Organization—ISO—and/or the American National Standards Institute—ANSI) continues to be a major impetus for improvement of programming language portability. Unfortunately, most compiler designers succumb to a compelling urge to provide "better" but nonstandard features for a standardized language. If portability is a critical requirement, source code must be restricted to the ISO or ANSI standard, even if other features exist.

*Availability of development tools* can shorten the time required to generate source code and can improve the quality of the code. Many programming languages may be acquired with a suite of tools that include debugging compilers, source code formatting aids, built-in editing facilities, tools for source code control, extensive subprogram libraries in a variety of application areas, browsers, cross-compilers for microprocessor development, macroprocessor capabilities, reverse engineering tools, and others. In fact, the concept of a good "software development environment" (e.g., [BAR84], [BEN89]) that includes both conventional and automated tools has been recognized as a key contributor to successful software engineering.

*Maintainability* of source code is critically important for all nontrivial software development efforts. Maintenance cannot be accomplished until software is understood. Earlier elements of the software configuration (i.e., design documentation) provide a foundation for understanding, but ultimately source code must be read and modified according to changes in design. Ease of design to code translation is an important element in source code maintainability. In addition, self-documenting characteristics of a language (e.g., allowable length of identifiers, labeling format, data type/structure definition) have a strong influence on maintainability.

### 16.2.4 Choosing a Language

The choice of a programming language for a specific project must take into account both engineering and psychological characteristics. However, the problem associated with choice may be moot if only one language is available or dictated by a requester. Meek [MEE80, p. 37] suggests a general philosophy when a programming language must be chosen:

> ...the art of choosing a language is to start with the problem, decide what its requirements are, and their relative importance, since it will probably be impossible to satisfy them all equally well (with a single language)...available languages should be measured against a list of requirements....

Among the criteria that are applied during an evaluation of available languages are (1) general application area, (2) algorithmic and computational complexity, (3) environment in which software will execute, (4) performance considerations, (5) data structure complexity, (6) knowledge of software development staff, and (7) availability of a good compiler or cross-compiler. Applications area of a project is a criterion that is applied most often during language selection. As we noted in Chapter 1, a number of major software application areas have evolved and de facto standard languages may be selected for each.

C is often the language of choice for the development of systems software, while languages such as Ada, C, and Modula-2 (along with FORTRAN and assembly language) are encountered in real-time applications. COBOL is the language for business applications, but the increasing use of fourth-generation languages has displaced it from its preeminent position. In the engineering/scientific area, FORTRAN remains the predominant language (although ALGOL, PL/1, PASCAL, and C have wide usage). Embedded software applications make use of the same languages applied in systems and real-time applications. The predominant language for personal computer users remains BASIC, but that language is rarely used by the developers of personal computer software products—more likely choices are PASCAL or C. Artificial intelligence applications make use of languages such as LISP, PROLOG, or OPS5, although other more conventional programming languages are used as well.

The rush toward object-oriented software development across most application domains has spawned many new languages and conventional language dialects. The most widely used object-oriented programming languages are Smalltalk, C++, and Objective-C. But languages such as Eiffel, Object-PASCAL, Flavors and many others are also used by growing numbers of software engineers.

The proliferation of "new and better" programming languages continues. Although many of these languages are attractive, it is sometimes better to choose a "weaker" (old) language that has solid documentation and support software, is familiar to everyone on the software development team, and has been successfully applied in the past. However, new languages should be thoroughly evaluated and the transition from old to new should occur, recognizing the psychological resistance to change that is encountered in all organizations.

### 16.2.5 Programming Languages and Software Engineering

Regardless of the software engineering paradigm, programming language will have impact on project planning, analysis, design, coding, testing, and maintenance. But the role of a programming language must be kept in perspective. Languages do provide the means for human-to-machine translation; however, the quality of the end result is more closely tied to the software engineering activities that precede and follow coding.

During project planning, a consideration of the technical characteristics of a programming language is rarely undertaken. However, planning for support tools associated with resource definition may require that a specific compiler (and associated software) or programming environment be specified. Cost and schedule estimation may require learning-curve adjustments because of staff inexperience with a language.

Once software requirements have been established,[1] the technical characteristics of candidate programming languages become more important. If complex data structures are required, languages with sophisticated data structure support (e.g., PASCAL and others) would merit careful evaluation. If high-performance, real-time capability is paramount, a language designed for real-time application (e.g., Ada) or memory-speed efficiency (e.g., C) might be specified. If many output reports and heavy file manipulation are specified, languages like COBOL or RPG might fit the bill. Ideally, software requirements should precipitate the selection of a language that best fits the processing to be accomplished. In practice, however, a language is often selected because "it's the only one we have running on our computer!"

The quality of a software design is established in a manner that is independent of programming language characteristics (a notable exception is object-oriented design, Chapter 12). However, language attributes do play a role in the quality of an implemented design and affect (both consciously and unconsciously) the way that design is specified.

In Chapter 10 we discussed a number of qualitative and quantitative measures of good design. The concepts of modularity and module independence were emphasized. Technical characteristics of many programming languages can affect these design concepts during the implementation of the design. To illustrate, consider the following examples:

> *Modularity* is supported by nearly all modern programming languages. COBOL, for example, supports a hierarchy of functions that integrates various levels of procedural abstraction (Chapter 10) with the modularity concept. The hierarchy consists of divisions, sections, paragraphs, sen-

[1]It is important to note that the language to be used for implementation can dictate the requirements analysis method that is chosen. For example, the use of an object-oriented language such as C++ might lead an analyst to choose OOA (Chapter 8) as the requirements analysis method. The converse is, of course, also true.

tences, and finally words. Each of these terms has a precise meaning in the language and helps to emphasize a modular implementation.

*Module independence* can be enhanced or subverted by language characteristics. For example, the Ada *package* supports the concept of information hiding while the use of internal procedures in PL/1 can lead to extensive global data that increase module coupling.

Data design (discussed in Chapters 8 and 10) can also be influenced by language characteristics. Programming languages such as Ada, C++, and Smalltalk support the concept of *abstract data types*—an important tool in data design and specification. Other more common languages, such as PASCAL, allow the definition of user-defined data types and the direct implementation of linked lists and other data structures. These features provide the designer with greater latitude during the preliminary and detail design steps.

In some cases, design requirements can only be satisfied when a language has special characteristics. Per Brinch-Hansen [HAN78] describes a set of language characteristics essential for implementation of a design that specifies distributed processes that are executing concurrently and must communicate and coordinate with one another. Languages such as concurrent PASCAL, Ada, or Modula-2 can be used to satisfy such designs.

The effect of programming language characteristics on the steps that comprise software testing is difficult to assess. Languages that directly support the structured constructs tend to reduce the cyclomatic complexity (Chapter 17) of a program, thereby making it somewhat easier to test. Languages that support the specification of external subprograms and procedures (e.g., FORTRAN) make integration testing much less error-prone. On the other hand, some technical characteristics of a language can impede testing. For example, block structuring in ALGOL can be specified in a manner that causes the loss of intermediate data when exit from a block occurs, thereby making the status of a program more difficult to assess.

Like testing, the effect of programming language characteristics on software maintenance is not fully understood. There is no question, however, that technical characteristics that enhance code readability and reduce complexity are important for effective maintenance. Further discussion of software maintenance is postponed until Chapter 20.

## 16.3 PROGRAMMING LANGUAGE FUNDAMENTALS

The technical characteristics of programming languages span an enormous number of topics that range from theoretical (e.g., formal language theory and specification) to pragmatic (e.g., functional comparisons of specific languages). In this section, a brief discussion of programming language fundamentals is presented. For more detailed discussions of programming language technology, the reader should reference Pratt [PRA84] or Sebesta [SEB89].

For the purposes of our discussion, programming language fundamentals will be presented within the context of four broad topics: data typing, subprogram mechanisms, control structures, and support for object-oriented approaches. All programming languages can be characterized with respect to these topics and the overall quality of a specific programming language can be judged with regard to the strengths and weakness related to each topic.

### 16.3.1 Data Types and Data Typing

Today, the merits of a modern programming language are judged by more than the syntax and breadth of its procedural constructs. *Data typing*, and the specific data types supported by a programming language, are an important aspect of language quality.

Pratt [PRA84] describes data types and data typing as "...a class of data objects together with a set of operations for creating and manipulating them." A data object inherits a set of fundamental attributes of the data type to which it belongs. A data object can take on a value that resides within the range of legitimate values for the data type and can be manipulated by operations that apply to the data type.

Simple data types span a wide range that includes *numeric* types (e.g., integer, complex, floating-point numbers), *enumeration* types (e.g., user-defined data types found in PASCAL), *boolean* types (e.g., true or false), and *string* types (e.g., alphanumeric data). More complex data types encompass data structures that run the gamut of simple one-dimensional arrays (vectors) to list structures to complex heterogeneous arrays and records.

The operations that may be performed on a particular data type and the manner in which different types may be manipulated in the same statement is controlled by *type checking* implemented within the programming language compiler or interpreter. Fairley [FAI85] defines five levels of type checking that are commonly encountered in programming languages:

Level 0: typeless
Level 1: automatic type coercion
Level 2: mixed mode
Level 3: pseudostrong type checking
Level 4: strong type checking

*Typeless* programming languages have no explicit means for data typing and, therefore, do not enforce type checking. Languages[2] such as BASIC, APL, LISP, and even COBOL fall into this category. Although each language does enable the user to define data structures, the representation of data contained within each data object is predefined.

[2]References to programming languages in this chapter assume "typical" or ANSI standard implementations. It is entirely possible that other versions of a language may exhibit characteristics that contradict our discussion.

*Automatic-type coercion* is a type checking mechanism that allows the programmer to mix different data types, but then converts operands of incompatible types, thus allowing requested operations to occur. For example, PL/1 assigns a numeric value of 0 to the boolean value *false* and a numeric value of 1 to the boolean value *true*. Hence arithmetic operations (normally applied to numeric data types) can be applied to boolean data types in PL/1.

*Mixed mode-type conversion* is similar in many respects to automatic-type coercion. Different data types within the same type category (e.g., two different numeric types) are converted to a single target type so that a specified operation can occur. FORTRAN's mixed-mode arithmetic (a feature that is best avoided) enables integer and real numbers to be used in a single programming language statement.

*Strong-type checking* occurs in programming languages that will only permit operations to be performed on data objects that are of the same prespecified data type. Operators, operands, and subprogram (module) interfaces are checked for type compatibility at compile time, at load time, and at run time. Ada compilers perform strong-type checking.

*Pseudostrong-type checking* has all of the characteristics of strong-type checking but is implemented in a manner that provides one or more loopholes [FAI85]. For example, although PASCAL checks interface compatibility within a single compiled program, it does not do so for separately compiled procedures (modules)—hence, there is a loophole in the enforcement of strong-type checking.

### 16.3.2 Subprograms

A *subprogram* is a separately compilable program component that contains a data and control structure. Throughout this book, we have referred to a *module* as a generic manifestation of a subprogram. Depending on the programming language, a subprogram may be called a subroutine, a procedure, a function, or any of a number of specialized names. Regardless of its name, the subprogram exhibits a set of generic characteristics: (1) a specification section that includes its name and interface description; (2) an implementation section that includes data and control structure; (3) an *activation mechanism* that enables the subprogram to be invoked from elsewhere in the program.

In conventional programming languages, each subprogram is an entity in itself, operating on data in a manner that is dictated by a larger program's control structure. In object-oriented programming languages, the classic view of the subprogram is replaced with the *object*.

### 16.3.3 Control Structures

At a fundamental level, all modern programming languages enable the programmer to represent sequence, condition, and repetition—the structured programming logical constructs. Most modern languages provide a syntax for direct specification of *if-then-else, do-while,* and *repeat-until* (as well as *case*). Other languages, such as LISP and APL, require the programmer to emulate the constructs within the syntax bounds of the language.

In addition to the basic procedural constructs of structured programming, other control structures may be present. *Recursion* creates a second activation of a subprogram during the first activation. That is, the subprogram invokes or activates itself as part of the defined procedure. *Concurrency* provides support for the creation of multiple tasks, the synchronization of tasks, and general communication between tasks. This language feature is invaluable when real-time or systems applications are undertaken. *Exception handling* is a programming language feature that traps user-defined or system error conditions and passes control to an exception handler for processing.

### 16.3.4 Support for Object-Oriented Approaches

In theory, the creation of objects and the construction of object-oriented software can be accomplished using any conventional programming language (e.g., C or PASCAL). But in practice, support for object-oriented approaches should be built directly into the programming language that will be used to implement an object-oriented design.

The fundamental concepts that underlie object-oriented programming were presented in Chapters 8 and 12. In addition to the features discussed in Sections 16.3.1 through 16.3.3, an object-oriented programming language should provide direct support for class definitions, inheritance, encapsulation, and messaging. In addition to these basic object-oriented constructs, many object-oriented languages implement additional features, such as multiple inheritance and polymorphism (different objects can receive messages with the same name).

The definition of classes is basic to an object-oriented approach. An object-oriented programming language defines a class name and specifies the private and public components of the class. To illustrate the general form of the object-oriented constructs, we use C++. A class, called **counter**, can be defined in the following manner [WEI88]:

```
class counter
{
    private:
        unsigned int value;
    public:
        counter ( );
        void increment ( );
        void decrement ( );
        unsigned int access_value ( );
};
```

A new class can be derived from the basic class definition in the following manner:

```
class special_counter : public counter
{
   a copy of the private data of counter
   private:
      data private to special_counter
      ...
};
```

Objects derived from the class special_counter can use all methods defined for the "parent class" counter. The definition of a class encapsulates data abstractions and the program components (methods) that operate on them. In C++, messages are sent to an object (an instantiation of a class) using the form

```
object_name.message (arguments);
```

where object_name identifies the object and message (arguments) describes the message to be sent to it.

The implementation details and the terminology for class definitions, inheritance, encapsulation, and messaging will vary from language to language. For example, the Smalltalk programming language [PIN88] defines a class using the following form:

- *Definition*—identifies the class
- *Private data*—attributes whose values are private to individual instances of the class
- *Shared data*—attributes whose values are shared by all instances of the class
- *Pool data*—attributes whose values are shared across multiple classes
- *Instance methods*—the procedures that implement messages that can be sent to an instance of a class
- *Class methods*—the procedures that implement messages that can be sent to a class (e.g., initialize shared data)

Although these terms differ from the C++ definition, the fundamental concept of the class remains unchanged. Similarly, inheritance, encapsulation, and messaging are implemented with a different syntax, but the same fundamental semantics. Each construct will be available in any language that is truly object-oriented.

## 16.4 LANGUAGE CLASSES

There are hundreds of programming languages that have been applied at one time or another to serious software development efforts. Even a detailed discussion of the five most common languages is beyond the scope of this book. The reader is referred to Pratt [PRA84] and Sebesta [SEB89] for thorough surveys and comparisons of the most common programming languages. In this section, four generations of programming languages are described and representative languages from each generation are discussed.

Any categorization of programming languages is open to debate. In many cases, one language might legitimately reside in more than one category. For the purposes of this book, we develop a set of language generations that correspond roughly to the historical evolution of programming languages. Figure 16.1 illustrates this categorization.

### 16.4.1 First-Generation Languages

The first language generation harkens back to the days of machine-level coding. Yet, some work with first-generation languages continues to this date. Machine code and its more human-readable equivalent—assembler language—represent the first language generation. These machine-dependent languages exhibit the lowest level of abstraction with which a program can be represented.

There are as many assembler languages as there are processor architectures with custom instruction sets. From a software engineering viewpoint,

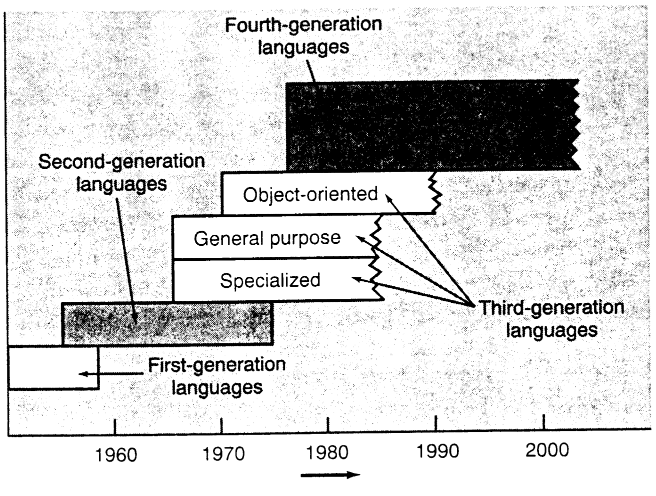

Fourth-generation languages
Second-generation languages
Object-oriented
General purpose
Specialized
Third-generation languages
First-generation languages
1960
1970
1980
1990
2000

**FIGURE 16.1.** Programming language generations.

such languages should be used only when a high-order language cannot meet requirements or is not supported.

### 16.4.2 Second-Generation Languages

Second-generation languages were developed in the late 1950s and early 1960s and serve as the foundation for all modern (third-generation) programming languages. Second-generation languages are characterized by broad usage, enormous software libraries, and the widest familiarity and acceptance. There is little debate that FORTRAN, COBOL, ALGOL, and (to some extent) BASIC are foundation languages by virtue of their maturity and acceptance.

FORTRAN has withstood 30 years of criticism to remain the premier programming language in engineering/scientific work.[3] The original standardized version of FORTRAN (called "FORTRAN-66") provided a powerful tool for computational problem solving, but lacked direct support of the structured constructs, had poor data typing, could not easily support string handling, and had many other deficiencies. The newer ANSI standard (called "FORTRAN-77") and the forthcoming standard correct some of the deficiencies found in earlier versions of the language. In many cases FORTRAN has been force-fit into application areas for which it was never designed and much of the criticism of the language has been somewhat unfair. For number crunching applications, FORTRAN remains the language of choice, but for system, real-time, or embedded product software applications, other languages provide compelling advantages.

COBOL, like FORTRAN, has reached maturity and is the accepted "standard" language for commercial data processing applications. Although the language is sometimes criticized for a lack of compactness, it has excellent data definition capabilities, is largely self-documenting, and provides support for a wide range of procedural techniques relevant to business data processing.

ALGOL is the forerunner of many third-generation languages and offers an extremely rich repertoire of procedural and data typing constructs. ALGOL has been used extensively in Europe, but has found little support (with the exception of academic environments) in the United States. The most commonly used version of the language, correctly termed "ALGOL-60," has been extended to a more powerful implementation, ALGOL-68. Both versions of the language support the notion of block structuring, dynamic storage allocation, recursion, and other characteristics that have had a strong influence on the modern languages that have followed.

[3]FORTRAN's dominance in engineering applications is starting to wain. Ada, C, and, to a lesser extent, PASCAL are replacing FORTRAN in many application domains. However, FORTRAN will be used widely well into the twenty-first century.

BASIC is a language that was originally designed to teach programming in a time-sharing mode. The language was moving toward obsolescence in the early 1970s, but has experienced a rebirth with the advent of personal computer systems. There are hundreds of versions of BASIC, making it difficult to discuss the benefits and deficiencies of the language.

### 16.4.3 Third-Generation Languages

Third-generation languages (also called *modern* or *structured programming languages*) are characterized by strong procedural and data structuring capabilities. The languages in this class can be divided into three broad categories, *general-purpose high-order languages, object-oriented high-order languages*, and *specialized languages*. All general-purpose and object-oriented high-order languages exhibit the technical characteristics discussed in Section 16.3. Specialized languages, on the other hand, have been designed to satisfy special requirements and have a syntax and form that are often unique.

**General-Purpose High-Order Languages** The earliest general-purpose high-order language (also a foundation language), ALGOL, served as a model for other languages in this category. Its descendents, PL/1, PASCAL, Modula-2, C, and Ada are being adopted as languages with the potential for broad spectrum applications (i.e., for use in engineering/scientific, embedded products, commercial, and/or systems application areas).

PL/1 might more properly be categorized as a 2.5-generation language. It was the first true broad-spectrum language, developed with a wide range of features that enable it to be used in many different application areas. PL/1 provides support for conventional engineering/scientific and business applications while at the same time enabling specification of sophisticated data structures, multitasking, complex I/O, list processing, and many other features. Subsets of the language have been developed to teach programming (PL/C), for use in microprocessor applications (PL/M), and for systems programming (PL/S).

PASCAL is a modern programming language that was developed in the early 1970s as a language for teaching modern techniques (e.g., structured programming) in software development. Since its introduction, PASCAL has found growing support from a broad audience of software developers and is used widely for engineering/scientific applications and systems programming (the language has been called "the FORTRAN of the 1980s"). PASCAL is a direct descendent of ALGOL and contains many of the same features: block structuring, strong data typing, direct support for recursion, and other complementary features. It has been implemented on computers of all sizes.

Modula-2 is an evolutionary outgrowth of PASCAL and (some would say) a possible alternative to the Ada programming language. Modula-2 couples direct implementation of design features such as information hiding, abstraction, and strong data typing with control structures to support recursion and concurrency. To date, the use of Modula-2 for industry applications has been limited.

The C programming language was originally developed as a language for operating system implementers. The UNIX operating system is implemented in C. Today, however, a vast array of software products, embedded applications, and systems software has been built using the C language. C was developed for the sophisticated software engineer and contains powerful features that give it considerable flexibility. These same features can also create problems. Cox [COX85] provides a poetic description of the language:

> One of my favorite hobbies is green wood working. A project begins not in the lumber yard with Kiln dried wood, but in the forest. A straight-grained oak tree is cut down, and worked into rustic furniture with an awe-inspiring assortment of antique tools....
>
> The tools of this hobby have a lot in common with the tools I use as a programmer. For example, the adze is a heavy blade on a four foot handle (like a hoe). It is a specialized tool, whose primary function is smoothing the rough surfaces of a split log. It is swung double-handed, standing astraddle the work. The razor sharp blade removes six-inch slabs of solid oak with a single blow, scant inches from unprotected feet and legs!
>
> I love that adze the way I love the C language. It is not a tool for fools and children. But in the hands of a skilled craftsman, it is capable of powerful, yet delicate work. Its potential for grave harm is so obvious that the danger provides the only safety mechanism; a healthy respect for what careless use can do!

Like other languages in this category, C supports sophisticated data structures and has reasonable typing characteristics, makes extensive use of pointers, and has a rich set of operators for computation and data manipulation. In addition, it enables the programmer to "get close to the machine" by providing assembly language-like features.

Ada was originally developed as a new standard language for embedded real-time computer systems to be developed for the U.S. Department of Defense. Today, the language is used widely in both defense and nondefense applications. Pascal-like in structure and notation (but far more powerful and complex), Ada supports a rich set of features that include multitasking, interrupt handling, intertask synchronization, and communication as well as a set of unique features such as the Ada *package*. Ada has created and continues to generate much controversy. Adherents praise its rich language structure and the focus on the Ada environment for software engineering rather than language-related esoterica. Opponents worry about the complexity of the language, the current inefficiency of operational compilers, and the long learning curve. It appears, however, that the benefits of the language will win out and that Ada may well dominate some application domains during the 1990s.

**Object-Oriented Languages** Object-oriented programming languages enable a software engineer to implement analysis and design models created using OOA and OOD (Chapters 8 and 12). These languages have characteristics described in Section 16.4.3.

Although dozens of object-oriented languages have been introduced over the past decade, only a few have gained any significant foothold in the marketplace: dialects of C (e.g., C++, Objective-C), Smalltalk, and Eiffel. Smalltalk, a "foundation" object-oriented language, was originally developed in the early 1970s to explore object-oriented concepts. Today, versions of Smalltalk are available on computers of all types, although the use of the language for the development of products and industry quality systems is limited. Object-oriented dialects of C have gained widespread use throughout the UNIX community and with many first-time developers of object-oriented systems. Building on the strengths of C, object-oriented dialects enable a smooth transition from this widely used, general-purpose, high-order language. Eiffel [MEY88] is one of a number of "new" object-oriented languages that are robust enough for industry application. Like C dialects and Smalltalk, Eiffel provides direct support for class definitions, inheritance, encapsulation, and messaging.

**Specialized Languages** Specialized languages are characterized by unusual syntactic forms that have been especially designed for a distinct application. Hundreds of specialized languages are in use today. In general, such languages have a much smaller user base than general-purpose languages. Among the languages that have found application within the software engineering community are LISP, PROLOG, APL, and FORTH.

LISP is a language that is especially suited to symbol manipulation and the list processing encountered in combinatorial problems. Used almost exclusively by the artificial intelligence community, the language is particularly well suited to theorem proving, tree searches, and other problem-solving activities. Subprograms are implemented as functions that make heavy use of recursion. Because each LISP function is a standalone entity, reusability can be achieved by creating libraries of primitive functions. In recent years, LISP has been used to develop a wide array of expert systems and expert system "compilers." LISP makes it relatively easy to specify facts, rules, and the corresponding inferences (implemented as LISP functions) that are required for knowledge-based systems.

PROLOG is another programming language that has found widespread use in the construction of expert systems. Like LISP, PROLOG provides features that support knowledge representation. Within the language, a uniform data structure, called the *term*, is used to construct all data and all programs. Each program consists of a set of clauses that represent facts, rules, and inferences. Both LISP and PROLOG are especially amenable to problems that treat objects and their relationships. For this reason, some people refer to LISP and PROLOG as object-oriented languages. In addition, the object-oriented nature of LISP and PROLOG enables each to be applied within the context of the prototyping paradigm for software engineering.

APL is an extremely concise and powerful language for array and vector manipulation. The language contains little support for structured constructs or data typing. APL does provide a rich set of computational operators and has gained a small but avid following for mathematical problem solving.

FORTH is a language designed for microprocessor software development. The language supports the definition of user-defined functions [implemented with post-fix (reverse-Polish) notation] that are executed in a stack-oriented manner for speed and memory efficiency.

From a software engineering standpoint, specialized languages provide both advantages and disadvantages. Because a specialized language has been designed to address a specific application, the translation of requirements to design to code implementation can be facilitated. On the other hand, most specialized languages are far less portable and often less maintainable than general-purpose languages.

### 16.4.4 Fourth-Generation Languages

Throughout the history of software development, we have attempted to generate computer programs at higher and higher levels of abstraction. First-generation programming languages worked at the machine instruction set level, the lowest possible level of abstraction. Second- and third-generation programming languages have raised the level at which we represent computer programs, but distinct and completely detailed algorithmic procedures still have to be specified. Over the past decade, *fourth-generation languages* (4GLs) have raised the level of abstraction still higher.

Fourth-generation languages, like all artificial languages, contain a distinct syntax for control and data structure representation. A 4GL, however, represents these structures at a higher level of abstraction by eliminating the need to specify algorithmic detail. For example, the statement:

```
COMPUTE NET-PRESENT-VALUE AND RETURN-ON-INVESTMENT FOR
EXPENDITURES #5 AND #9.
```

is typical of a 4GL statement. The 4GL system "knows" how to compute the desired financial data and does so without requiring the software developer to specify the appropriate algorithms. It should be apparent that the "knowledge" described above is *domain-specific*. That is, the same 4GL would undoubtedly choke on:

```
COMPUTE THE ROOTS OF TRANSCENDENTAL EQUATION #3 AND APPLY
THEM TO THE PHYSICAL MODEL.
```

although another 4GL, designed specifically for the application domain implied above, might do the job nicely.

Fourth-generation languages combine *procedural* and *nonprocedural* characteristics. That is, the language enables the user to specify conditions and corresponding actions (the procedural component) while at the same time encouraging the user to indicate the desired outcome (the nonprocedural component) and then applying its domain-specific knowledge to fill in the procedural details.

Martin ([MAR85], [MAR86]) presents a comprehensive discussion of 4GLs and develops the following broad categories.

**Query Languages** To date, the vast majority of 4GLs have been developed for use in conjunction with database applications. Such query languages enable the user to manipulate information contained in a pre-existing database in a sophisticated manner. Some query languages require a complex syntax that is no simpler (and in some cases, worse) than a third-generation language. For example [MAR85]:

```
list by region (87.act.sep.sales)
sum (87.est.sep.sales), (sum (sum (87.act.sep.sales)
```

However, other query languages available today offer a natural language interface that allows the user to state [INT86]:

```
For the eastern and western regions, how did actual sales for last
month compare with forecasts?
```

Needless to say, the second approach would be favored by most users.

**Program Generators** Program generators represent another, somewhat more sophisticated, class of 4GLs. Rather than relying on a predefined database as its point of focus, a program generator enables the user to create complete third-generation language programs using (many claim) an order-of-magnitude fewer statements. These *very-high-level programming languages* make heavy use of procedural and data abstractions (Chapter 10). Unfortunately for those working in the engineered products and systems domain, most program generators available today focus exclusively on business information systems applications and generate programs in COBOL. However, a new generation of CASE tools enables a software engineer to model an engineering application graphically and then generate C or Ada source code from the graphical model.

**Other 4GLS** Although query languages and application generators are the most common 4GLs, other categories exist. *Decision support languages* enable "nonprogrammers" to perform a variety of *what-if* analyses that range from simple two-dimensional spreadsheet models to sophisticated statistical or operations research modeling systems. *Prototyping languages* have been developed to assist in creating prototypes by facilitating the creation of user interfaces and dialogs and providing a means for data modeling. *Formal specification languages* (discussed in Chapter 9) can be considered to be

4GLs when such languages produce machine-executable software. Finally, tools used in a personal computer environment (e.g., spreadsheets, database systems, Macintosh Hypercard) enable the user to "program" at a higher level of abstraction than previously available.

## 16.5 CODING STYLE

After source code is generated, the function of a module should be apparent without reference to a design specification. In other words, code must be understandable. Coding style encompasses a coding philosophy that stresses simplicity and clarity. In their landmark text on the subject, Kernighan and Plauger [KER78, p. 9] state:

> Writing a computer program eventually boils down to writing a sequence of statements in the language at hand. How each of those statements is expressed determines in large measure the intelligibility of the whole....

The elements of style include internal (source code level) documentation, methods for data declaration, an approach to statement construction, and techniques for I/O. In the sections that follow, we consider each of these topics.

### 16.5.1 Code Documentation

Internal documentation of source code begins with the selection of identifier (variables and labels) names, continues with the placement and composition of commenting, and concludes with the visual organization of the program.

The selection of meaningful identifier names is crucial to understanding. Languages that limit variable names or labels to only a few characters inherently obscure meaning. Consider the following three statements:

```
D=V*T
DIST= HORVEL*TIME
DISTANCE= HORIZONTAL.VELOCITY * TIME.TRAVELED.IN.SECS;
```

The BASIC language expression is undeniably concise, but the meaning of D = V*T is unclear unless the reader has prior information. The FORTRAN expression provides more information, but the meaning of DIST and HORVEL could be misinterpreted. The ALGOL statement leaves little doubt regarding the meaning of the calculation. These statements illustrate the way in which identifiers may be chosen to help document code.

It can be argued that "wordy" expressions (like the ALGOL statement above) obscure logical flow and make modification difficult. Obviously, common sense must be applied when identifiers are selected. Unnecessarily long identifiers do indeed provide a potential for error (not to mention a backache from sitting long hours typing at a workstation). Studies [SHN80] indicate, however, that even for small programs meaningful identifiers improve comprehension. In terms of the syntactic/semantic model discussed in Section 16.2.2, meaningful names "simplify the conversion from program syntax to internal semantic structure" [SHN80].

The ability to express natural language comments as part of a source code listing is provided by all general-purpose programming languages. However, certain questions arise:

- How many comments are "enough"?
- Where should the comments be placed?
- Do comments obscure logic flow?
- Can comments mislead the reader?
- Are comments "unmaintainable," and therefore unreliable?

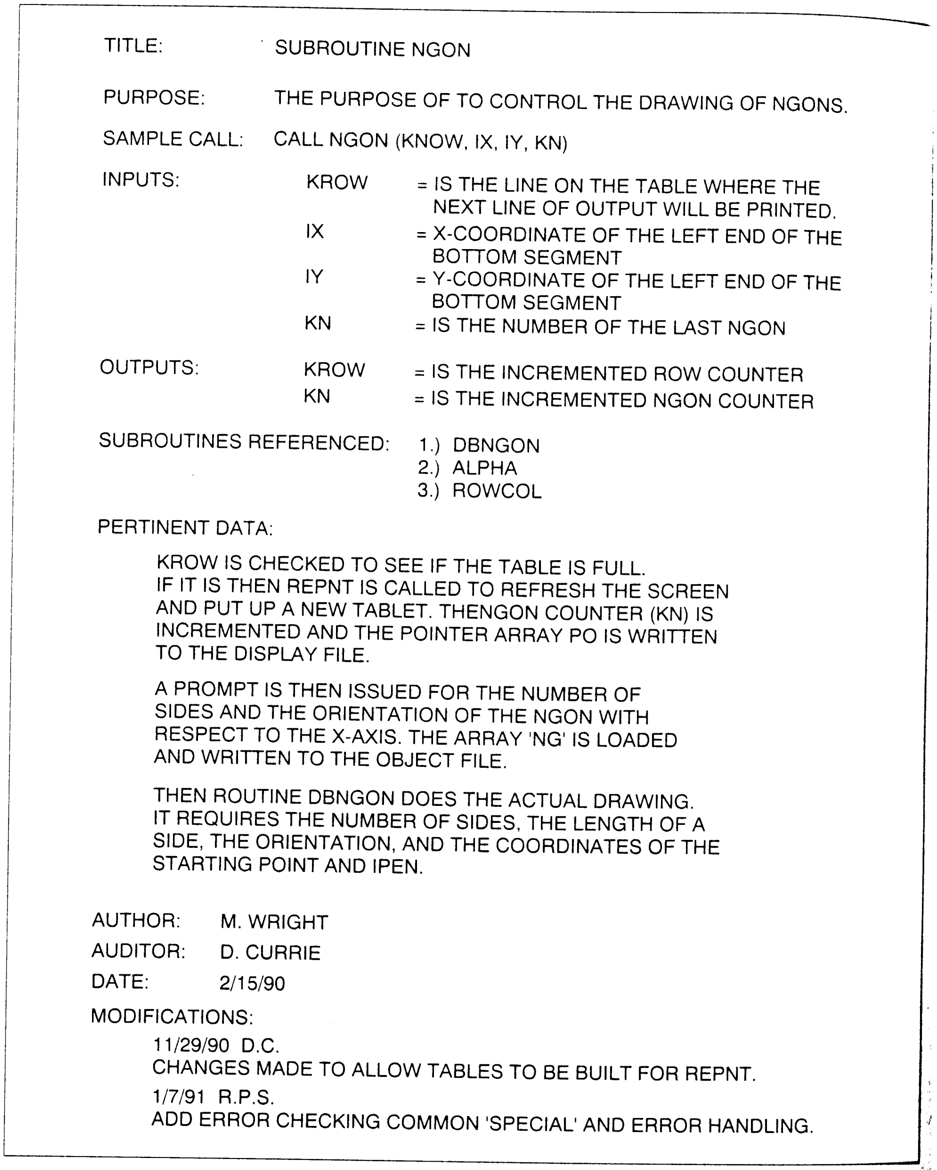
There are few definitive answers to the above questions. But one thing is clear: *Software must contain internal documentation*. Comments provide the developer with one means of communicating with other readers of the source code. Comments can provide a clear guide to understanding during the last phase of software engineering—maintenance.

There are many guidelines that have been proposed for commenting. Prologue comments and functional comments are two categories that require somewhat different approaches. *Prologue comments* should appear at the beginning of every module. The format for such comments is:

**1.** A statement of purpose that indicates the function of the module
**2.** An interface description that includes
  **a.** a sample "calling sequence"
  **b.** a description of all arguments
  **c.** a list of all subordinate modules
**3.** A discussion of pertinent data such as important variables and their use restrictions and limitations, and other important information
**4.** A development history that includes
  **a.** the name of the module designer (author)
  **b.** the name of the reviewer (auditor) and date
  **c.** modification dates and description

An example of prologue comments is given in Figure 16.2.

Descriptive comments are embedded within the body of source code and are used to describe processing functions. A primary guideline for such commenting is expressed by VanTassel [VAN78]: "comments should provide



```
TITLE:           SUBROUTINE NGON

PURPOSE:         THE PURPOSE OF TO CONTROL THE DRAWING OF NGONS.

SAMPLE CALL:     CALL NGON (KNOW, IX, IY, KN)

INPUTS:               KROW      = IS THE LINE ON THE TABLE WHERE THE
                                  NEXT LINE OF OUTPUT WILL BE PRINTED.
                      IX        = X-COORDINATE OF THE LEFT END OF THE
                                  BOTTOM SEGMENT
                      IY        = Y-COORDINATE OF THE LEFT END OF THE
                                  BOTTOM SEGMENT
                      KN        = IS THE NUMBER OF THE LAST NGON

OUTPUTS:              KROW      = IS THE INCREMENTED ROW COUNTER
                      KN        = IS THE INCREMENTED NGON COUNTER

SUBROUTINES REFERENCED:    1.) DBNGON
                           2.) ALPHA
                           3.) ROWCOL

PERTINENT DATA:

     KROW IS CHECKED TO SEE IF THE TABLE IS FULL.
     IF IT IS THEN REPNT IS CALLED TO REFRESH THE SCREEN
     AND PUT UP A NEW TABLET. THENGON COUNTER (KN) IS
     INCREMENTED AND THE POINTER ARRAY PO IS WRITTEN
     TO THE DISPLAY FILE.

     A PROMPT IS THEN ISSUED FOR THE NUMBER OF
     SIDES AND THE ORIENTATION OF THE NGON WITH
     RESPECT TO THE X-AXIS. THE ARRAY 'NG' IS LOADED
     AND WRITTEN TO THE OBJECT FILE.

     THEN ROUTINE DBNGON DOES THE ACTUAL DRAWING.
     IT REQUIRES THE NUMBER OF SIDES, THE LENGTH OF A
     SIDE, THE ORIENTATION, AND THE COORDINATES OF THE
     STARTING POINT AND IPEN.

AUTHOR:      M. WRIGHT
AUDITOR:     D. CURRIE
DATE:        2/15/90
MODIFICATIONS:
     11/29/90  D.C.
     CHANGES MADE TO ALLOW TABLES TO BE BUILT FOR REPNT.
     1/7/91  R.P.S.
     ADD ERROR CHECKING COMMON 'SPECIAL' AND ERROR HANDLING.
```

**FIGURE 16.2.**
Code documentation.

something extra, not just paraphrase the code." In addition, descriptive comments should

- Describe blocks of code, rather than commenting every line.
- Use blank lines or indentation so that comments can be readily distinguished from code.
- Be correct; an incorrect or misleading comment is worse than no comment at all.

With proper identifier mnemonics and good commenting, adequate internal documentation is assured.

When a detailed procedural design is represented using a program design language (Chapter 10), design documentation can be embedded directly into the source listing as comment statements. This technique is particularly useful when implementation is to be done in assembler language and helps to ensure that both code and design will be maintained when changes are made to either.

The form of the source code as it appears on the listing is an important contributor to readability. Source code indentation indicates logical constructs and blocks of code by indenting from the left margin so that these attributes are visually offset. Like commenting, the best approach to indentation is open to debate. Manual indentation can become complicated as code modification occurs, and experiments [SHN80] indicate that only a marginal improvement in understanding accrues. Probably the best approach is to use an automatic code formatter (a CASE tool) that will properly indent source code. By eliminating the burden of indentation from the coder, form may be improved with relatively little effort.

### 16.5.2 Data Declaration

The complexity and organization of data structure are defined during the design step. The style of data declaration is established when code is generated. A number of relatively simple guidelines can be established to make data more understandable and maintenance simpler.

The order of data declarations should be standardized even if the programming language has no mandatory requirements. For example, declaration ordering for a FORTRAN module might be:

1. All explicit declarations (for high quality, all variables should be declared)

   INTEGER, REAL, DOUBLE PRECISION, . . .

2. All global data blocks

   COMMON/block-name/. . .

3. All local arrays

   DIMENSION array names and dimensions

4. All file declarations

   DEFINE FILE, OPEN, CLOSE

Ordering makes attributes easier to find, expediting testing, debugging, and maintenance.

When multiple variable names are declared with a single statement, an alphabetical ordering of names is worthwhile. Similarly, labeled global data (e.g., FORTRAN common blocks) should be ordered alphabetically.

If a complex data structure is prescribed by design, commenting should be used to explain peculiarities inherent in a programming language implementation. For example, a linked list data structure in C or a user-defined data type in PASCAL might require supplementary documentation contained in comments.

### 16.5.3 Statement Construction

The construction of software logical flow is established during design. The construction of individual statements, however, is part of the coding step. Statement construction should abide by one overriding rule: Each statement should be simple and direct; code should not be convoluted to effect efficiency.

Many programming languages allow multiple statements per line. The space saving aspects of this feature are hardly justified by the poor readability that results. Consider the following two code segments:

```
DO I = 1 TO N-1; T = I; DO J = I + 1 TO N; F A(J) < A(T) THEN DO T=J;
END;
IF T <> I THEN DO H=A(T); A(T)=A(I); A(I)=T; END; END;
```

The loop structure and conditional operations contained in the above segment are masked by the multistatement-per-line construction. Reorganizing the form of the code:

```
DO I = 1 TO N-1;
    T = I;
    DO J = I+1 TO N;
        IF A(J) < A(T) THEN DO
            T=J;
        END;
    IF T <> I THEN DO
        H=A(T);
        A(T)=A(I);
        A(I)=T;
        END;
    END;
```

Here, simple statement construction and indentation illuminates the logical and functional characteristics of the segment. Individual source code statements can be simplified by

- Avoiding the use of complicated conditional tests
- Eliminating tests on negative conditions
- Avoiding heavy nesting of loops or conditions
- Using parentheses to clarify logical or arithmetic expressions
- Using spacing and/or readability symbols to clarify statement content
- Using only ANSI standard features
- Thinking: Could I understand this if I was not the person who coded it?

Each of the above guidelines strives to "keep it simple."

### 16.5.4 Input/Output

The style of input and output is established during software requirements analysis and design, not coding. However, the manner in which I/O is implemented can be the determining characteristic for system acceptance by a user community. Input and output style will vary with the degree of human interaction. For batch-oriented I/O, logical input organization, meaningful input/output error checking, good I/O error recovery, and rational output report formats are desirable characteristics. For interactive I/O, a simple, guided input scheme, extensive error checking and recovery, human-engineered output, and consistency of I/O format become primary concerns.[4]

Regardless of the batch or interactive nature of software, a number of I/O style guidelines should be considered during design and coding:

- Validate all input data.
- Check the plausibility of important combinations of input items.
- Keep the input format simple.
- Use end-of-data indicators, rather than requiring a user to specify "number-of-items."
- Label interactive input requests, specifying available choices or bounding values.
- Keep the input format uniform when a programming language has stringent formatting requirements.
- Label all output and design all reports.

The style of I/O is affected by many other characteristics such as I/O devices (e.g., terminal or workstation type, computer graphics device, mouse, etc.), user sophistication, and communication environment.

[4]See Chapter 14 for additional guidelines on the design of human-computer interfaces.

## 16.6 EFFICIENCY

In well-engineered systems, there is a natural tendency to use critical re sources efficiently. Processor cycles and memory locations are often viewe as critical resources, and the coding step is seen as the last point where mi croseconds or bits can be squeezed out of the software. Although efficienc is a commendable goal, three maxims should be stated before we discuss th topic further. First, efficiency is a *performance requirement* and should therefore, be established during software requirements analysis. Softwar should be as efficient as is required, not as efficient as is humanly possible Second, efficiency is improved with good design. Third, code efficiency an code simplicity go hand in hand. In general, don't sacrifice clarity, readabil- ity, or correctness for nonessential improvements in efficiency.

### 16.6.1 Code Efficiency

The efficiency of source code is directly tied to the efficiency of algorithms defined during detail design. However, coding style can have an effect on execution speed and memory requirement. The following set of guidelines can always be applied when detail design is translated into code:

- Simplify arithmetic and logical expressions before committing to code.
- Carefully evaluate nested loops to determine if statements or expressions can be moved outside.
- When possible, avoid the use of multi-dimensional arrays.
- When possible, avoid the use of pointers and complex lists.
- Use "fast" arithmetic operations.
- Don't mix data types, even if the language allows it.
- Use integer arithmetic and boolean expressions, whenever possible.

Many compilers have optimizing features that automatically generate effi- cient code by collapsing repetitive expressions, performing loop evaluation, using fast arithmetic, and applying other efficiency-related algorithms. For applications in which efficiency is paramount, such compilers are an indis- pensable coding tool.

### 16.6.2 Memory Efficiency

Memory restrictions in the large machine ("mainframe") and workstation world are largely a thing of the past. Low-cost memory provides a large physical address space and virtual memory management provides applica- tion software with an enormous logical address space. Memory efficiency for such environments cannot be equated to minimum memory used. Rather, memory efficiency must take into account the "paging" characteristics of an operating system. In general, code locality or maintenance of functional domain via the structured constructs is an excellent method for reducing paging and thereby increasing efficiency.

Memory restrictions in the embedded microprocessor world are a very real concern, although low-cost, high-density memory is evolving rapidly. If minimal memory is demanded by system requirements (e.g., a high-volume, low-cost product), high-order language compilers must be carefully evaluated for memory compression feature, or, as a last resort, assembler language may have to be used.

Unlike many other system characteristics that must be traded against one another, techniques for execution time efficiency can sometimes lead to memory efficiency. For example, limiting the use of three- or four-dimensional arrays results in simple element access algorithms that are fast and short. Again, the key to memory efficiency is "keep it simple."

### 16.6.3 Input/Output Efficiency

Two classes of I/O should be considered when efficiency is discussed: I/O directed at a human or I/O directed to another device (e.g., a disk or another computer). Input supplied by a user and output produced for a user are efficient when the information can be supplied or understood with an economy of intellectual effort.

Efficiency of I/O to other hardware is an extremely complicated topic and is beyond the scope of this book. From the coding (and detail design) standpoint, however, a few simple guidelines that improve I/O efficiency can be stated:

- The number of I/O requests should be minimized.
- All I/O should be buffered to reduce communication overhead.
- For secondary memory (e.g., disk), the simplest acceptable access method should be selected and used.
- I/O to secondary memory devices should be blocked.
- I/O to terminals and printers should recognize features of the device that could improve quality or speed.
- Remember that "superefficient" I/O is worthless if it can't be understood.

As we noted earlier in this chapter, I/O design establishes style and ultimately dictates efficiency. The guidelines presented above are applicable to both design and coding steps of the software engineering process.

## 16.7 SUMMARY

The coding step of software engineering is a process of translation. Detail design is translated into a programming language that is ultimately (and automatically) transformed into machine-executable instructions. Psychological and technical characteristics of a programming language affect the ease of translation from design and the effort required to test and maintain software. These characteristics may be applied to programming languages that fall into one of four language generations.

Style is an important attribute of source code and can determine the intelligibility of a program. The elements of style include internal documentation, methods for data declaration, procedures for statement construction, and I/O coding techniques. In all cases, simplicity and clarity are key characteristics. An offshoot of coding style is the execution time and/or memory efficiency that are achieved. Although efficiency can be an extremely important requirement, we should remember that an "efficient" program that is unintelligible has questionable value.

Coding lies at the kernel of the software engineering process. Critically important steps have preceded coding, relegating it to a somewhat mechanistic translation of a detail design specification. Equally important steps follow coding, and it is a discussion of these steps and related topics that constitute the next part of this book.

## REFERENCES

[BAR84] Barstow, D. R., H. E. Shobe, and E. Sandewall, *Interactive Programming Environments,* McGraw-Hill, 1984.

[BEN89] Bennett, K. H., *Software Engineering Environments,* Halsted Press (Wiley), 1989.

[COX85] Cox, B., *Software ICs and Objective C,* Productivity Products International (now StepStone, Inc.), 1985.

[FAI85] Fairley, R. E., *Software Engineering Concepts,* McGraw-Hill, 1985.

[HAN78] Brinch-Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *CACM,* vol. 21, no. 11, November 1978.

[INT86] *Intellect System Documentation,* Artificial Intelligence Corporation, Waltham, MA, 1986.

[JEN74] Jensen, K., and N. Wirth, *Pascal User Manual and Report,* Springer-Verlag, 1974.

[KER78] Kernighan, B., and P. Plauger, *The Elements of Programming Style,* 2d ed., McGraw-Hill, 1978.

[KLA80] Klatzky, R., *Human Memory,* 2d ed., W. H. Freeman and Co., 1980.

[LED81] Ledgard, H., and M. Marcotty, *The Programming Language Landscape,* SRA, 1981.

[MEE80] Meek, B., and P. Heath (eds.), *Guide to Good Programming,* Halsted Press (Wiley), 1980.

[MAR85] Martin, J., *Fourth Generation Languages,* volume 1, Prentice-Hall, 1985.

[MAR86] Martin, J., and J. Leben, *Fourth Generation Languages,* volume 2, Prentice-Hall, 1986.

[MEY88] Meyer, B., *Object-Oriented Software Construction,* Prentice-Hall, 1988.

[PIN88] Pinson, L. J., and R. S. Weiner, *An Introduction to Object-Oriented Programming and Smalltalk,* Addison-Wesley, 1988.

[PRA84] Pratt, T., *Programming Languages,* 2d ed., Prentice-Hall, 1984.

[SEB89] Sebesta, R.W., *Concepts of Programming Languages,* Addison-Wesley, 1989.

[SHE81] Shepard, S., E. Kruesi, and B. Curtis, "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications," *Proc. 5th Intl. Conf. Software Engineering,* IEEE, San Diego, March 1981, pp. 207–214.

[SHN80] Shneiderman, B., *Software Psychology,* Winthrop Publishers, 1980.

[VAN78] VanTassel, D., *Program Style, Design, Efficiency, Debugging and Testing,* 2d ed., Prentice-Hall, 1978.

[WEI71] Weinberg, G., *The Psychology of Computer Programming,* Van Nostrand, 1971.

[WEI88] Weiner, R.S., and L. J. Pinson, *An Introduction to Object-Oriented Programming and C++,* Addison-Wesley, 1988.

## PROBLEMS AND POINTS TO PONDER

**16.1** Do some research on natural language processing (a book by Harris entitled *Natural Language* is a good starting point) and write a position paper on the probability of natural language programming.

**16.2** Much of the work in software psychology has centered on the characteristics of programming languages and their effects on the coding task. Write a paper that presents some of the more current work in this area.

**16.3** Select one or more programming languages and provide examples of each of the psychological characteristics (e.g., uniformity, ambiguity, etc.) discussed in Section 16.2.

**16.4** Select the one programming language that you feel best satisfies the software engineering traits that are discussed in Section 16.2.3. Would your choice change if the technical characteristics of the language were also considered?

**16.5** Select one of the third-generation languages discussed in Section 16.4.3. Prepare a brief summary of important language characteristics and write a small program that illustrates the language syntax.

**16.6** Select any specialized language and prepare a summary of its important characteristics and special features. Write a small program that illustrates its language syntax.

**16.7** Select any object-oriented language and prepare a summary of its important characteristics and special features. Write a small program that illustrates its language syntax.

**16.8** Select any fourth-generation language (see [MAR85]) and prepare a summary of its important characteristics and special features. Write a small "program" that illustrates its language syntax.

**16.9** Expert systems applications are one of a number of specialized application areas. Research the LISP and PROLOG languages and summarize their strengths and weaknesses in this application area. How are the languages similar? How do they differ?

**16.10** Ada is a programming language with a wide variety of features. How is Ada different from programming languages such as PASCAL or C? In providing your answer, focus on the three fundamental characteristics discussed in Section 16.3.

**16.11** Ada has been the source of much controversy over the last few years. Summarize the arguments (pro and con) that have been presented in the literature.

**16.12** List by priority those style guidelines that you feel are most important. Justify your selection. Are these guidelines language-dependent, i.e., do some languages obviate the need for a particular guideline?

**16.13 through 16.21** Code and attempt to implement the procedural designs for the correspondingly numbered problems in Chapter 10. You may use the programming language of your choice, but remember the style and clarity guidelines discussed in this chapter.

## FURTHER READINGS

Programming languages are fundamental to an understanding of computer science and should be understood individually and in relationship to one another. Books by Sebesta [SEB89], Pratt [PRA84], and Ledgard and Marcotty [LED81] satisfy both requirements nicely. A book by Smedema et al. (*The Programming Languages Pascal, Modula, CHILL and Ada,* Prentice-Hall, 1983) presents a thumbnail sketch of these important programming languages.

The *Elements of Programming Style* [KER78] remains *must* reading for all individuals who intend to generate source code. The authors have provided an extensive, annotated set of rules for coding (and design) that are well worth heeding. In addition Jon Bentley (*Programming Pearls,* Addison-Wesley, 1986; *More Programming Pearls,* Addison-Wesley, 1988) presents a worthwhile collection of style guidelines and clever language solutions to common programming problems. Books by Weiler (*The Programmers Craft,* Reston, 1983), Liffick (*The Software Development Source Book,* Addison-Wesley, 1985), and Ledgard (*Professional Software: Programming Concepts,* Addison-Wesley, 1987) provide additional information about style.

There is no "best" textbook that can be chosen from the hundreds that have been written about languages within any one of the language classes that we have discussed. The following list contains a representative sample of source material for many of the programming languages discussed in this chapter:

FORTRAN: Ellis, M., *A Structured Approach to FORTRAN 77 Programming,* Addison-Wesley, 1989.

COBOL: Johnson, B. M., and M. Ruwe, *Professional Programming in COBOL,* Prentice-Hall, 1990.

ALGOL: Brailsford, D., and Walker, A., *Introductory ALGOL-68 Programming,* Wiley, 1979.

BASIC: Pearson, O. R., *Programming with Basic,* McGraw-Hill, 1986.

PL/1: Tremblay, J. P., et al., *Structured PL/1 (PL/C) Programming,* McGraw-Hill, 1980.

PASCAL: Mallozzi, J.S., *Program Design in Pascal*, McGraw-Hill, 1989.
C: Hutchison, R., and S.B. Just, *Programming Using the C Language*, McGraw-Hill, 1988.
C++: Lippman, S.B., *A C++ Primer*, Addison-Wesley, 1989.
Ada: Cohen, N.H., *Ada as a Second Language*, McGraw-Hill, 1986.
Modula-2: Eisenbach, S., and C. Sadler, *Program Design with Modula-2*, Addison-Wesley, 1989.
LISP: Anderson, J.R., et al., *Essential Lisp*, Addison-Wesley, 1987.
OPS5: Sherman, P.D., and J.C. Martin, *An OPS5 Primer*, Prentice-Hall, 1990.
Prolog: Clocksin, W., and C. Mellish, *Programming in Prolog*, 2d ed., Springer-Verlag, 1984.
Smalltalk: Goldberg, A., and D. Robson, *Smalltalk-80*, Addison-Wesley, 1983.
FORTH: Katzen, H., *Invitation to FORTH*, Petrocelli, 1981.

An introduction to formal language theory can be found in *Jewels of Formal Language Theory* (A. Salomaa, Computer Science Press, 1981) and Meyer (*Introduction to the Theory of Programming Languages*, Prentice-Hall, 1988). These books survey morphic representations, formal syntax specification, DOL languages, and many other topics. An equally rigorous treatment of language formalism can also be found in Dijkstra and Feijen (*A Method of Programming*, Addison-Wesley, 1988).